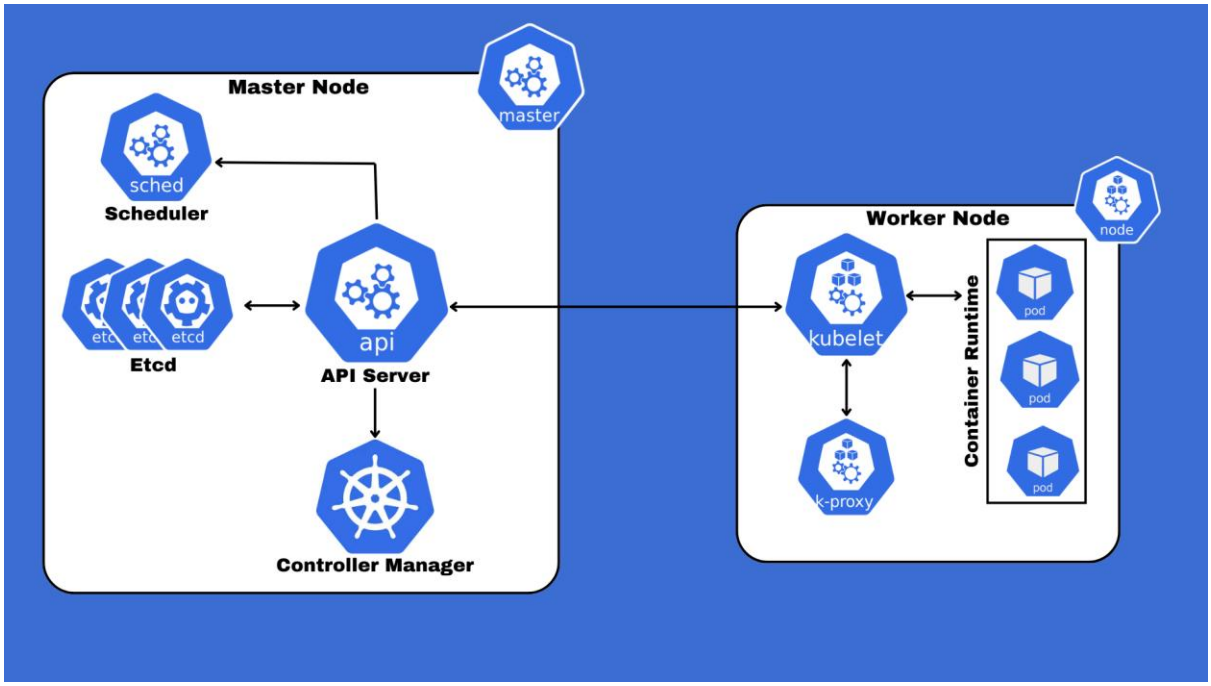
DEVOPS SHACK

KUBERNETES Architecture PART-2



Kubernetes is a powerful open-source platform for automating the deployment, scaling, and operation of containerized applications. It achieves this through its well-designed architecture, which is composed of several key components that work together to manage the lifecycle of applications. In this document, we'll explore the architecture of Kubernetes in detail, covering both the **Master Node** and **Worker Node** components, along with real-world examples.

Overview of Kubernetes Architecture

At a high level, Kubernetes follows a **master-worker** architecture, where:

- **Master nodes** are responsible for managing the entire cluster.

- **Worker nodes** are where the actual applications (in containers) run.

The Kubernetes architecture can be divided into several layers:

1. **Master Node** (Control Plane)
2. **Worker Node** (Data Plane)
3. **Additional Components**

1. Master Node (Control Plane)

The **Master Node** is the brain of the Kubernetes cluster, managing and orchestrating all worker nodes and containers running in the cluster. It handles tasks like scheduling, scaling, and monitoring. The main components of the Master Node are:

- **API Server**
- **etcd**
- **Scheduler**
- **Controller Manager**

1.1 API Server (kube-apiserver)

The **API Server** is the central access point for all administrative operations on the cluster. It exposes the Kubernetes API, which allows interaction with the cluster via HTTP. Both users (via kubectl) and internal components of Kubernetes (like the Scheduler and Controller Manager) interact with the API Server to perform tasks.

Key responsibilities:

- Handling all REST API requests for interacting with the cluster.
- Serving as the gateway to the cluster's control plane.
- Validating and configuring data for the API objects (like Pods, Services, etc.).

Real-world Example: When a developer runs the command kubectl apply -f deployment.yml, this request goes to the API Server. The API Server processes the request, stores the configuration in etcd, and communicates with the Scheduler to schedule the new Pods on available worker nodes.

1.2 etcd

etcd is a distributed key-value store that stores all cluster data. It is a highly available and consistent datastore that Kubernetes uses to store the current state and configuration of the cluster.

Key responsibilities:

- Storing cluster state and configuration.
- Maintaining consistency across the cluster.

- Acting as the source of truth for the cluster's state.

  Real-world Example: Whenever you deploy or delete a Pod, the current state and desired state of the cluster are updated in etcd. The API Server retrieves and updates cluster data from etcd to ensure the cluster is in its desired state.

  1.3 Scheduler (kube-scheduler)

  The **Scheduler** is responsible for scheduling pods onto available worker nodes based on resource availability, constraints, and policies. The Scheduler considers factors such as CPU, memory, affinity rules, taints, and tolerations to decide the best node for a new Pod.

  Key responsibilities:

- Assigning Pods to nodes based on resource availability.
- Ensuring balanced resource utilization across nodes.
- Respecting scheduling constraints (like node affinity and tolerations).

  Real-world Example: Let’s say you have a Pod definition where the Pod requires 1 CPU and 1GB memory. When you create the Pod, the Scheduler will identify a node that has enough resources to accommodate this Pod and schedules it accordingly.

  1.4 Controller Manager (kube-controller-manager)

  The **Controller Manager** runs a set of controllers that regulate the state of the cluster and ensure that the cluster’s desired state matches its actual state. Each controller is responsible for managing a specific aspect of the cluster.

  Key controllers:

- **Node Controller**: Monitors the health of nodes and takes action when nodes go down.
- **Replication Controller**: Ensures that the correct number of pod replicas are running at all times.
- **Endpoints Controller**: Populates the Endpoints object with Pod IP addresses.
- **Service Account Controller**: Manages service accounts and associated API tokens.

  Real-world Example: When a node goes down, the Node Controller detects this and informs the Scheduler and API Server. Then, the Replication Controller ensures that any lost Pods are recreated on healthy nodes.

2. Worker Node (Data Plane)

The **Worker Node** is where the actual applications (in the form of containers) are executed. A Kubernetes cluster can have multiple worker nodes, and the Master Node manages these worker nodes. Each worker node runs the following key components:

- **Kubelet**
- **Kube Proxy**
- **Container Runtime**

2.1 Kubelet

The **Kubelet** is the primary agent running on each worker node. It communicates with the API Server to ensure that containers are running as expected and that the node meets the cluster's desired state.

Key responsibilities:

- Communicating with the Master Node to retrieve Pod definitions.
- Managing the containers on its node and ensuring that they are healthy and running.
- Reporting the health and status of the node and its pods to the API Server.

Real-world Example: If the Master Node schedules a new Pod on a worker node, the Kubelet on that node will fetch the Pod's specification from the API Server and instruct the container runtime (like Docker) to start the containers.

2.2 Kube Proxy

Kube Proxy is responsible for networking in Kubernetes. It ensures that network traffic is properly routed between services and pods inside the cluster. It handles service discovery and load balancing for network requests.

Key responsibilities:

- Maintaining network rules to allow communication between Pods, Services, and external clients.
- Load balancing traffic across Pods in a Service.
- Forwarding traffic to appropriate backend Pods.

Real-world Example: When a service is created in Kubernetes, Kube Proxy ensures that any traffic directed at the service IP is forwarded to the appropriate Pods, thus enabling internal and external communication.

2.3 Container Runtime

The **Container Runtime** is the software responsible for running the actual containers in each Pod. Kubernetes supports different container runtimes, such as **Docker**, **containerd**, or **CRI-O**.

Key responsibilities:

- Running and managing containers in Pods.
- Handling container lifecycle operations such as start, stop, and restart.

Real-world Example: Let's assume your cluster uses Docker as its container runtime. When the Kubelet receives instructions from the API Server to start a Pod, it communicates with Docker to launch the container(s) as per the Pod specification.

---

3. Additional Components

In addition to the Master and Worker nodes, Kubernetes includes additional components that are integral to the cluster's functioning.

3.1 Pod

A **Pod** is the smallest, most basic deployable object in Kubernetes. It can contain one or more containers, and all containers in a Pod share the same network namespace and storage.

Key characteristics:

- Pods are ephemeral by design, and Kubernetes may restart them as needed.
- Pods are assigned unique IP addresses but can share storage volumes.

Real-world Example: If you deploy a web application in Kubernetes, a Pod will host the container(s) that run your application. All containers in that Pod will share the same network and storage, and they can communicate with each other directly via localhost.

3.2 Service

A **Service** is an abstraction that defines a logical set of Pods and provides a stable endpoint (IP and DNS) for accessing them. Services enable load balancing, service discovery, and decoupling between Pods and their consumers.

Key types of Services:

- **ClusterIP**: Exposes the service internally within the cluster.
- **NodePort**: Exposes the service on a static port on each node.
- **LoadBalancer**: Provisions an external load balancer to expose the service to external traffic.

Real-world Example: If you have a web application that needs to be accessible from the internet, you would create a **LoadBalancer** service that assigns an external IP to your application. Requests made to this external IP are routed to the Pods in the service.

3.3 Namespace

Namespaces provide a way to divide cluster resources between different users or groups. They are particularly useful in multi-tenant environments, as they allow for resource isolation and name collision prevention.

Real-world Example: In a large organization where multiple teams share the same Kubernetes cluster, you can create separate namespaces for each team, ensuring that resources are isolated and conflicts are minimized.

3.4 Ingress

An **Ingress** is an API object that manages external access to services within the cluster, typically HTTP or HTTPS traffic. It provides load balancing, SSL termination, and name-based virtual hosting.

Real-world Example: You might use an Ingress controller to route traffic to different applications running in the same cluster, such as routing www.myapp.com to one service and api.myapp.com to another.

3.5 Persistent Volumes (PV) and Persistent Volume Claims (PVC)

A **Persistent Volume (PV)** is a piece of storage in the cluster that has been provisioned by an administrator. A **Persistent Volume Claim (PVC)** is a request by a user for storage, which is fulfilled by binding the claim to a PV.

Real-world Example: If you have a database running in a Pod, you would create a PVC to request persistent storage. The Kubernetes cluster would bind this PVC to an available PV, allowing the database to store its data persistently across Pod restarts.

3.6 ConfigMap

A **ConfigMap** is a key-value store in Kubernetes used to pass configuration data into Pods without including it in the container image. ConfigMaps decouple configuration artifacts from the containerized applications, allowing you to manage configurations separately and change them without rebuilding container images.

Key responsibilities:

- Storing configuration data that Pods use, such as environment variables, command-line arguments, or configuration files.
- Providing flexibility to update configurations without redeploying the containers.

Real-world Example: If your application requires database connection settings (e.g., host, port), you can store these settings in a ConfigMap. The Pod can reference the ConfigMap and inject the configuration values as environment variables or mount them as files inside the container.

3.7 Secret

A **Secret** is similar to a ConfigMap but is specifically designed to store sensitive data, such as passwords, API keys, and tokens. Kubernetes provides encryption mechanisms to secure this data at rest and ensures that sensitive information is not exposed in the cluster.

Key responsibilities:

- Storing sensitive data securely.
- Providing access control mechanisms to ensure only authorized Pods can access the secrets.

Real-world Example: If your application needs to connect to an external API using an API key, you can store this key in a Kubernetes Secret. The Secret can then be injected into the Pod as an environment variable or mounted as a file.

3.8 Volume

A **Volume** in Kubernetes allows data to persist even if the Pod or container is terminated. Kubernetes supports various types of volumes, including **emptyDir**, **hostPath**, **nfs**, **persistentVolumeClaim**, and more. Volumes ensure that data is not lost during Pod rescheduling or container restarts.

Key responsibilities:

- Providing storage that can persist beyond the lifecycle of individual Pods.
- Supporting different storage backends (e.g., cloud storage, NFS, or local disk).

Real-world Example: For a database Pod that stores its data locally, you can define a PersistentVolume and PersistentVolumeClaim. The database's data will persist on the underlying storage even if the Pod is restarted or moved to another node.

---

4. Kubernetes Controllers

Kubernetes controllers are control loops that constantly compare the desired state of the cluster to its actual state and make the necessary changes to bring the actual state closer to the desired state.

4.1 ReplicaSet Controller

A **ReplicaSet** is a Kubernetes controller that ensures a specified number of Pod replicas are running at any given time. If a Pod goes down or is deleted, the ReplicaSet controller will automatically create a new Pod to maintain the desired state.

Key responsibilities:

- Maintaining a stable set of replicas for a Pod.
- Ensuring high availability by restarting Pods if they fail or get terminated.

Real-world Example: If you define a ReplicaSet with replicas: 3, Kubernetes will ensure that three instances of your application are always running. If one Pod fails, the ReplicaSet will automatically start a new Pod to replace it.

4.2 Deployment Controller

A **Deployment** is a higher-level controller that manages ReplicaSets and provides additional functionality such as rolling updates and rollbacks. The Deployment controller ensures that updates to Pods are rolled out in a controlled fashion and allows you to roll back to a previous version in case of failure.

Key responsibilities:

- Managing ReplicaSets to ensure the correct number of Pods are running.
- Supporting rolling updates to update Pods without downtime.

- Facilitating rollbacks to a previous version if something goes wrong during an update.

  Real-world Example: When you update the container image of an application in a Deployment, Kubernetes will update the Pods gradually (rolling update) to ensure minimal downtime. If the new image has a problem, you can rollback to the previous image.

  4.3 StatefulSet Controller

  A **StatefulSet** is used for managing stateful applications, such as databases, where the identity and order of Pods matter. It ensures that Pods are created in a specific order and retain persistent storage across restarts.

  Key responsibilities:

- Maintaining the identity of each Pod, ensuring that each Pod gets the same persistent storage volume.
- Ensuring Pods are created, updated, or deleted in a specific order.

  Real-world Example: For applications like Cassandra or MySQL, where the state (data) of each instance must persist, a StatefulSet ensures that each instance gets its own volume and that instances are started and terminated in the correct order.

  4.4 DaemonSet Controller

  A **DaemonSet** ensures that a specific Pod runs on all (or a subset of) nodes in the cluster. This is particularly useful for running cluster-wide services like logging, monitoring, or network proxies.

  Key responsibilities:

- Ensuring a Pod is running on every node (or a specified set of nodes) in the cluster.
- Running background services like monitoring agents or log collectors on each node.

  Real-world Example: If you are using Prometheus to monitor your cluster, you can create a DaemonSet that runs a Prometheus exporter on every node to collect metrics. Similarly, a log collector like Fluentd can be deployed using a DaemonSet to gather logs from every node.

  4.5 Job and CronJob Controllers

  A **Job** in Kubernetes is used to run batch processes or tasks that run to completion. Once the job is completed successfully, the Pod is terminated. A **CronJob** is a special type of Job that runs at scheduled intervals (like a cron job in Unix).

  Key responsibilities:

- Managing batch jobs that run until completion.
- Scheduling and managing recurring jobs (CronJobs).

  Real-world Example: If you need to perform a daily data backup, you can define a CronJob that runs a backup Pod every day at midnight. The CronJob controller will ensure that the backup task runs according to the schedule.

5. Kubernetes Networking

Kubernetes provides a powerful and flexible networking model that allows communication between Pods, services, and external clients.

5.1 Pod Networking

Kubernetes assumes that every Pod in a cluster can communicate with every other Pod, regardless of the node it is running on. This is achieved using the **CNI (Container Network Interface)** model, which allows different network plugins (e.g., Flannel, Calico, Weave) to provide networking capabilities.

Key responsibilities:

- Allowing Pods to communicate with each other across nodes.
- Assigning IP addresses to Pods for direct communication.

Real-world Example: If your web application Pod needs to communicate with a backend database Pod, Kubernetes ensures that both Pods can talk to each other over the cluster network, regardless of their physical location in the cluster.

5.2 Service Networking

Kubernetes services provide a stable endpoint for Pods, allowing communication between services and external clients. **ClusterIP**, **NodePort**, and **LoadBalancer** services enable internal and external communication within the cluster.

Real-world Example: A frontend application might expose itself using a LoadBalancer service, which allows external traffic to reach the application. The backend database might expose itself as a ClusterIP service, allowing only internal traffic from other Pods.

5.3 Ingress

An **Ingress** controller is responsible for routing external HTTP/HTTPS traffic to the appropriate service in the cluster. It provides a single point of entry and can handle SSL termination, path-based routing, and virtual hosts.

Real-world Example: If you have multiple services running (e.g., an API and a web app), you can define an Ingress resource that routes requests to api.example.com to the API service and requests to www.example.com to the web app service.

Conclusion

Kubernetes is a powerful and comprehensive platform for managing containerized applications in a highly scalable and automated way. Its architecture consists of master and worker nodes, each playing specific roles in managing, deploying, and scaling applications. Components like the API Server, etcd, Kubelet, and controllers work together to maintain the desired state of the system while automating many operational tasks such as scaling, failover, and load balancing.